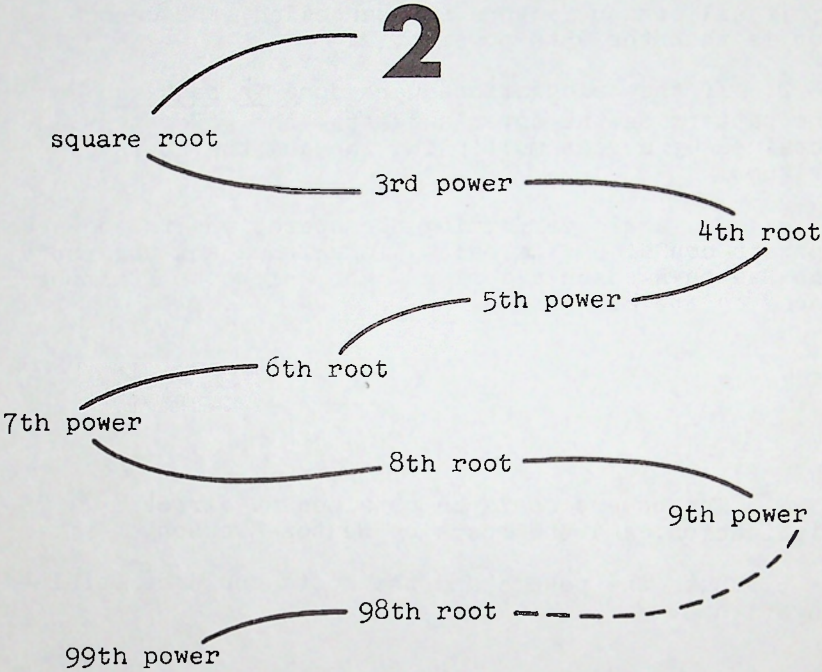
VOLUME 3 NUMBER 11 NOVEMBER 1975 32

Popular Computing



$$2^{\frac{3\quad 5\quad 7\quad 9\quad 11\quad 13\quad 15\quad 17\quad \cdots\quad 99}{2\quad 4\quad 6\quad 8\quad 10\quad 12\quad 14\quad 16\quad \cdots\quad 98}}$$

Error Amplification

Error Amplification

A calculation is given on the cover, involving an exponent on 2 having 49 odd factors in its numerator and 49 even factors in its denominator. There are many ways in which this calculation could be made:

1. Working first just with the exponent, its value can be calculated, and a log-antilog operation can be performed on that result. Much preliminary cancellation on the long fraction could shorten this work; if all common factors are cancelled, the denominator is then the 95th power of 2.

2. If the calculations were done in sequence, as the pattern on the cover suggests, the powers could be obtained by direct multiplication and the roots by logarithms.

3. Or, again performing the operations in sequence, the powers could be obtained by logarithms and the roots by the Newton-Raphson algorithm, for which the Kth root is found by iterating on

$$x_{n+1} = \frac{1}{K}\left[\frac{N}{x_n^{K-1}} + (K-1)x_n\right]$$

4. The powers could be obtained by direct multiplication, and the roots by Newton-Raphson.

5. Both the powers and the roots could be obtained by logarithms.

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $18 per year, or $15 if remittance accompanies the order. For Canada and Mexico, add $4 per year to the above rates. For all other countries, add $6 per year to the above rates. Back issues $2 each.

For all five ways, all the calculations could be performed to various degrees of precision.

Question: will these various ways lead to the same result? Obviously not; the real question is to what extent the results differ due to cascading of the errors in each calculation.

Method 1 was performed with 100-digit precision, yielding the result:

The exponent: 7.958923738717876149812705024217046
140293154042473332135734787051717
376016313210129737854003906249841

The result: 248.8139801957822733667554023591555
046685584290316749319866406018138
313952749538506720022824777795902

This result can be considered the "true" value, to at least 90 significant digits. Taking this value as a reference standard, some other results are as follows:

Method 1, EXACMATH 25-digit precision:

248.8139801957822733667201

Method 5, Fortran double precision:

248.81398019578227336669250

Method 4, Fortran double precision:

248.8139801957822733667648

Method 4, Fortran single precision:

248.81397405

Further results are solicited, using the other methods, other programming languages, and other levels of precision.

(The "true" result and the first result above were calculated by David Babcock; the last three results above were calculated by Dorothy Cady.) □

Double or Take

In issue 35 (April 1975) of Games & Puzzles, and again in issue 39 (August 1975) there appeared the game of Double or Take, a two-player number game. One player selects a positive integer. The opponent may either double it, or subtract a square, or subtract a cube. The winner is the one who reaches zero.

For the integers from 1 to 11, the following analysis indicates the status:

The player presented with		
1	Takes 1 and	wins
2	Can take only 1, leaving 1, or can double to 4; either way	loses
3	Takes 1, leaving opponent 2, and	wins
4	Takes 4 and	wins
5	Can take 1 or 4, leaving 4 or 1, or can double to 10, after which opponent takes 8, and hence	loses
6	Takes 1, leaving 5	wins
7	Can take 1 or 4, leaving 6 or 3, or double to 14, after which opponent takes 9, leaving 5, and hence	loses
8	Takes 1, reducing to case 7, and	wins
9	Takes 9 and	wins
10	Takes 8, reducing to case 2	wins
11	Takes 4, reducing to case 7	wins

By working one's way up, it can be shown that 17 and 19 are losing positions, as is 50.

Which brings us to case 12. The player presented with 12 cannot subtract, since each possibility leads to a win by the opponent, according to the analysis above. The other alternative is to double to 24. We then have:

If the opponent now takes 1, the player can take 16 and win.
If the opponent now takes 8, the player can take 16 and win.
If the opponent now takes 9, the player can take 8 and win.
If the opponent now takes 16, the player can take 8 and win.

None of these choices is appealing to the opponent, assuming that he desires to win himself. The opponent has two more choices: he can double again to 48, or he can take 4, leaving the player 20. So what choices does a player have when presented with 20?

Take 1, leaving 19 (**)	
Take 4, leaving 16	a clear loss.
Take 8, leaving 12 (*)	
Take 9, leaving 11	a clear loss.
Take 16, leaving 4	a clear loss.
Or double to 40 (**)	

And things now get sticky. In the case marked (*), the case 12 has been handed back to the opponent, and the circle could go on indefinitely. In the cases marked (**), the problem has been pushed on to still larger numbers which have not yet been analyzed. To quote from the article in Games & Puzzles:

> "...it does suggest that there may be many indecisive numbers which are neither winning nor losing; a player receiving one will change it into another indecisive number and failure to do so will lose him the game. No single definitely indecisive number has yet been determined, let alone, for example, a closed sequence of indecisive numbers."

We have then four possible situations for which a computer program might provide solutions:

(1) Extend the list of definite losing numbers that begins 2, 5, 7, 17, 19,...

(2) Extend the list of indecisive numbers that begins 12, 24, 48,...

(3) Extend the list of definite winning numbers.

(4) Produce a program to play the game against a human player. The program should select a starting integer at random between 100 and 999, with the first move to be made by the human player.

Learning by Doing

In a second, or intermediate, course in computing, the student should be lured (coerced, dragooned) into working on a term project, assigned at the start of the semester and due at final exam time. This should be a computer problem, preferably of the students' own choice, done in workmanlike manner to demonstrate that the student has learned something of the computing art. The final result should be packaged neatly, to include:

1. An English statement of the problem.

2. Flowcharts of the logic of the solution (or the equivalent of flowcharts, if the student prefers some other way of expressing his logic).

3. Listings of the program.

4. Results, clearly labelled.

5. A test procedure and test results.

6. Conclusions (what was learned from the work; what would be done differently if the project were to be repeated; limitations on the results; suggestions for further research, etc.).

The packaged term project should be saved for use in eventual job interviews.

Experience has shown that the student's chief problem is that of selecting the problem he intends to work on. In all likelihood, he has so far worked only on problems that were assigned (and hence clearly labelled computer problems), for which much of the analysis has already been done for him.

There is usually a fair amount of panic at the time this assignment is made, since it puts the student on his own for the first time. About a third of a typical class will suggest one of the following:

1. "May I use a problem I did in my Fortran class last semester?" No, that problem was defined and analyzed for you, and the object now is to see how you do with a new problem. Further, you did that problem; it's time for you to do another one.

2. "I've been assigned to a big problem at work; may I turn it in for this project?" No; that problem concerns your work; try an isolated problem here, one that you can deal with thoroughly and completely. (And experience has shown that when this restriction is relaxed, the end result always seems to be someone else's work, and the student can barely explain what went on.)

3. "I don't know where to find a problem to work on." Well, there are systematic collections of good problems; you might try browsing through Problems for Computer Solution (Gruenberger and Jaffray, Wiley, 1965) which outlines some 90 problems that would be suitable. The best problem is the one that interests you.

4. "I'm a business (music, chemistry,..., mathematics) major; I'll do a problem in business (music, chemistry,...,mathematics)."

There's the real problem. What is needed, early on in the semester, is a proposal by the student of just what he intends to do, so that he can be saved from the extremes of plunging into a problem that is either trivial or too grandiose. In the former case, he will produce something that requires little or no knowledge of computing; in the latter case, he will look sad at final exam time (when the computing center is saturated) and wail that he needs just one more run of his program.

The phrase "a business problem" is rather vague. A new attack on Bill of Materials scheduling? An inventory control program for 10,000 line items? A table of base pay times hours worked? Or what?

The trouble is, of course, that the whole idea is brand new to most students; he has never been placed in the awkward position of making a selection that is within his own capabilities (indeed, he has probably never been told what constitutes a decent computer problem). Further, he has never had to define a problem and outline an attack on it; this has been done for him for all of the 14 years he has been in school. It helps if he can see samples of what this is all about, so a collection of old term projects (both good ones and bad ones) should be made available to him. It would help even more if he could be shown some sample proposals. A few are given here.

I

A comparison will be made between the Gauss-Seidel and Gauss elimination algorithms for the solution of simultaneous linear equations. A number of sets of six such equations will be constructed with known roots. Programs will be written in Fortran to solve such systems with the two algorithms, both in single and double precision. The solutions will be compared for the following:

1. Computation time.
2. Accuracy of the results.
3. Compilation time.

One of the sets of equations will involve a singular matrix, to determine how this condition is handled by the programs.

II

A small inventory of 25 items will be set up, and daily changes to that inventory will serve as input to a program. The program is to update the inventory, and print a report showing, for each line item, the quantity on hand, items out of stock, reorder conditions, lead time to arrival of new stock, and those items requiring expediting. For the small inventory involved, all results will be hand calculated as a test of the logic.

III

The COBOL reference manual at our installation lists 58 error messages for errors that can occur at compile time. A program will be written that will trigger each of these error messages.

IV

The melody of a song can be expressed as a series of numbers. The pitch of each note can be expressed by numbering the notes of the scale, and the duration of the note can be coded on a scale from 1 to 8. With a given melody so coded, an algorithm can be applied to it to translate it into a new melody. The simplest such algorithm would be to reflect each note around a middle value, to transform high notes into low notes and vice versa, leaving the duration of each note fixed.

Several algorithms will be devised, and applied to ten "standard" tunes. The resulting translated melodies will be played, and the most pleasing result will be recorded and submitted on a cassette as part of this project.

V

The Los Angeles Times prints about 500 column-inches of text each day in its main news section. Some of these inches can be identified as politically oriented:

A. Favorable to Republicans.
B. Unfavorable to Republicans.
C. Favorable to Democrats.
D. Unfavorable to Democrats.
E. Favorable to third-party candidates.
F. Unfavorable to third-party candidates.
G. Completely neutral.

In theory, material that reflects the political leanings of the newspaper's editors should be confined to the editorial pages, and the news pages should be unbiassed. In practice, the amount of text space allotted to a candidate or a party reflects the paper's views, however unconsciously.

The pages of each issue for the 8 weeks preceding the last general election will be examined, and a listing made of the column-inches in each of the first four categories given above, as objectively as possible. Ratios will be calculated of the following:

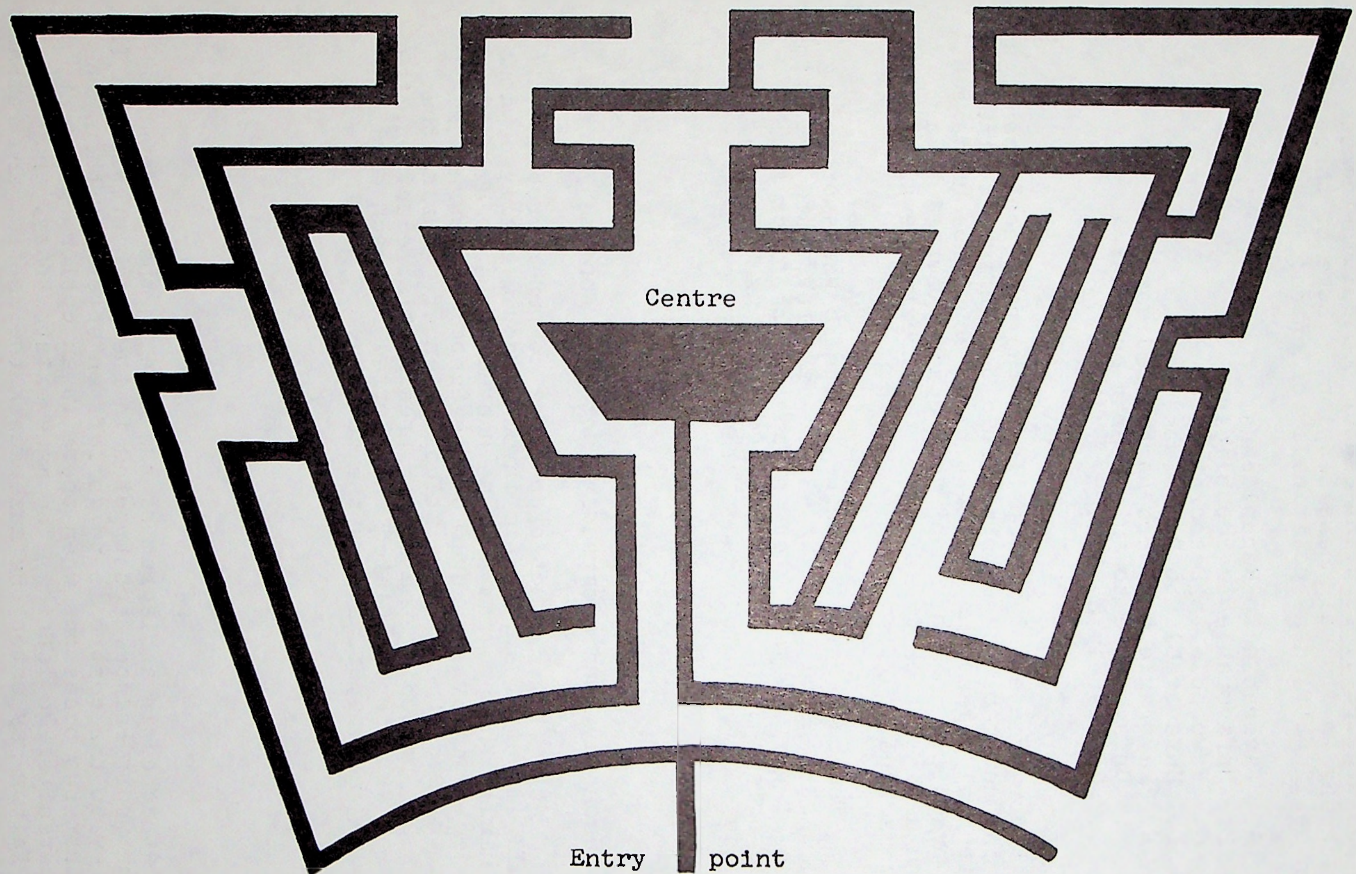
$$\frac{A}{B} \qquad \frac{C}{D} \qquad \frac{A}{C} \qquad \frac{A+D}{B+C}$$

for each day, and progressive totalled for the 56 days.

While this is not properly a computer problem (the small amount of arithmetic involved could easily be done by hand), the program will be useful for a much larger project jointly sponsored by the School of Journalism and the Department of Political Science. During the 12 weeks preceding the coming presidential election, the ratios will be calculated and plotted for each of 15 leading newspapers across the country.

VI

Attached is a diagram of the maze in the gardens at Hampton Court Palace, constructed in the reign of William III. "The key to the centre is to go left on entering, then, on the first two occasions when there is an option, go right, but thereafter go left." The maze exists today, and for 2p a visitor may enter the maze, seek the centre, and retrace his way out.



Plan of the maze at Hampton Court Palace, England. The path to be followed is the black line.

Given a new maze, described in terms of the coordinates of the branch points, a program is to be written to explore the maze and output the directions for the choices to be made to proceed from the entry point either to the center or to a specified exit. The choices will be limited to two at each branch point.

One test of the program will be the reproduction of the directions quoted above for the Hampton Court maze. The computer solution for the other mazes will be checked by hand.

VII

If the numbers from 1 to 20 are permuted, what is the distribution of runs up and runs down of the numbers? Consider the following possible permutations:

A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
B	1	20	2	19	3	18	4	17	5	16	6	15	7	14	8	13	9	12	10	11
C	20	18	16	14	1	3	5	7	12	10	8	6	4	2	9	11	19	17	15	13
D	2	12	17	4	13	10	1	5	15	19	20	7	16	14	6	18	3	8	9	11
E	11	19	2	7	12	1	8	6	18	13	15	20	17	10	16	14	3	4	5	9
F	1	15	14	2	16	13	3	17	12	4	18	11	5	19	10	6	20	9	7	8

For A, there is just one run up, and none down, but for permutations taken at random, the chance of arrangement A is just 1 in 20!, or about 1 in 2.4 times 10 to the 18th power.

For B, there are 10 runs up and 9 down, but this, too, is an unlikely arrangement.

For C, there are two runs up and three down.

For D, there are six runs up and five down.

For E, there are seven runs up and six down.

For F, there are seven runs up and six down.

The complete distribution of all possibilities for all 20! permutations could be determined by theory. I propose to approximate the distribution by sampling random permutations and counting the runs.

In order to do this, I will need a random number generator and an algorithm for forming random permutations. For the former, I will use the generator described on page 8 of issue 21 of POPULAR COMPUTING (my work will be done in Fortran). For the latter, one of the following schemes will be used:

1. Generate an array of 40 numbers. In the even positions put the numbers from 1 to 20. In the odd positions, generate 20 random numbers. Then sort the 20 pairs, using the random numbers as the sort key. Although it is inefficient, I will bubble sort the 20 pairs. After sorting, the right hand number in each pair of numbers will be a random permutation of the numbers from 1 to 20.

2. Use the random number generator to generate integers in the range from 1 to 20, and let these integers be the subscripts for entries in an array of dimension 20. First zero out this array. Select elements in the array by the choice of subscript. If the chosen element is zero, fill it with the next consecutive integer, starting with 1. If the element is not zero (i.e., it has already been selected), proceed to another element. When all the elements are filled, the array contains the desired random permutation.

3. In the above scheme, the first 16 or so elements will be filled rapidly (that is, the condition of duplicates will not occur too often), but the last 4 or so may take an undue amount of time. A variation might be tried. Apply the scheme described until 16 elements are filled. Then insert the remaining four numbers into the four blank slots, picking one of 24 arrangements of those four positions at random, again using the random number generator.

At least 1000 random permutations will be generated, and a distribution made of the lengths of the runs. This distribution will be compared to the theoretical (if I can obtain that). The program will be generalized so that it can operate on dimensions other than 20. If time permits, runs will be made on permutations of 10 and 30 items.

VIII

I would like to try to calculate the number (15000!). (I understand that the current record for factorials is 10000!, and that all the factorials by thousands are on file.) Even if I don't succeed in obtaining the desired result, I believe that working on the project will be worthwhile, and I will be able to at least provide hints and suggestions for the next person who tries to break the record.

Since this calculation will consume considerable amounts of machine time, I propose to calculate the following test data first:

1. The exact number of expected digits.
2. The number of low order zeros.
3. Some of the high order digits, and some of the low order non-zero digits.

In addition, I will use my program to calculate (and check with known results) 500!, 1000!, and 10000! before requesting a commitment of machine time for the long run.

I believe that I can hold intermediate results in storage by packing six decimal digits per machine word. I will need packing and unpacking subroutines, and a subroutine for decimal multiplication. I will test these subroutines before making any long runs.

IX

Problem H5 in Problems for Computer Solution calls for the creation of abstract paintings by a computer program. The program is to select the size and shape of various geometric figures and their position on a canvas. I wish to explore this notion extensively with the aid of the plotter now available, which should aid greatly in the mechanical chore of examining the program's results. The plotter will allow up to three primary colors for the figures, and the choice of color will also be made by the program. The plotter routines also permit the figures to appear in outline form, or solid (filled in), as well as various degrees of cross-hatching.

It is stated in Problem H5 that an important aspect of the problem is the determination of when to stop. I propose to put this decision into the program in terms of the area covered by the random figures. This will be a bit tricky, since the figures overlap. However, if the total area covered by the figures is limited to some fraction of the available area, with or without overlap, and this limit is a parameter of the program, then the program will be able to output finished art without intervention in any quantity.

X

The ratio of successive terms of the Fibonacci sequence approximates the golden mean:

$$\frac{1+\sqrt{5}}{2} = 1.6180339887498948482045\ldots$$

Thus, we see

$$144/89 = \underline{1.617}9775$$

$$1597/987 = \underline{1.61803}44$$

$$121393/75025 = \underline{1.618033988}6$$

Using the EXACMATH package, I propose to write a program to generate successive terms of the Fibonacci sequence, form the ratio, and determine to how many digits the ratio agrees with the golden mean. The limits of the EXACMATH package will let me carry these calculations to at least the 1000th term of the sequence. My output will be a table of values (term number against the number of digits of agreement) and a graph showing the rate of growth of the function being explored.

XI

The Raindrop Problem, which appeared in issue 6 of POPULAR COMPUTING, called for selecting a point at random in the unit square as the center of a circle whose radius is taken at random between zero and 1/2 unit. The problem asked for the number of such circles needed to completely cover the unit square.

A crude solution will be attempted by subdividing the square into 400 smaller squares. A mathematical test can be devised to determine whether or not each of these smaller squares is covered by one of the circles. The results of 100 trials will be plotted, to obtain an approximation to the desired distribution. For a few of these trials, the method will be repeated with the large square subdivided into 900 smaller squares, to determine if the method could lead to a correct solution. □

A Problem in Strategy

--and our 3rd contest

A random number generator is available that outputs 3-digit integers uniformly distributed in the range 000 through 999. The following game is to be played. The generator will be called ten times. From the ten random numbers, five are to be selected having the smallest variance (the variance being calculated by the formula

$$\frac{5(\text{sum of squares}) - (\text{sum})^2}{25}\Bigg)$$

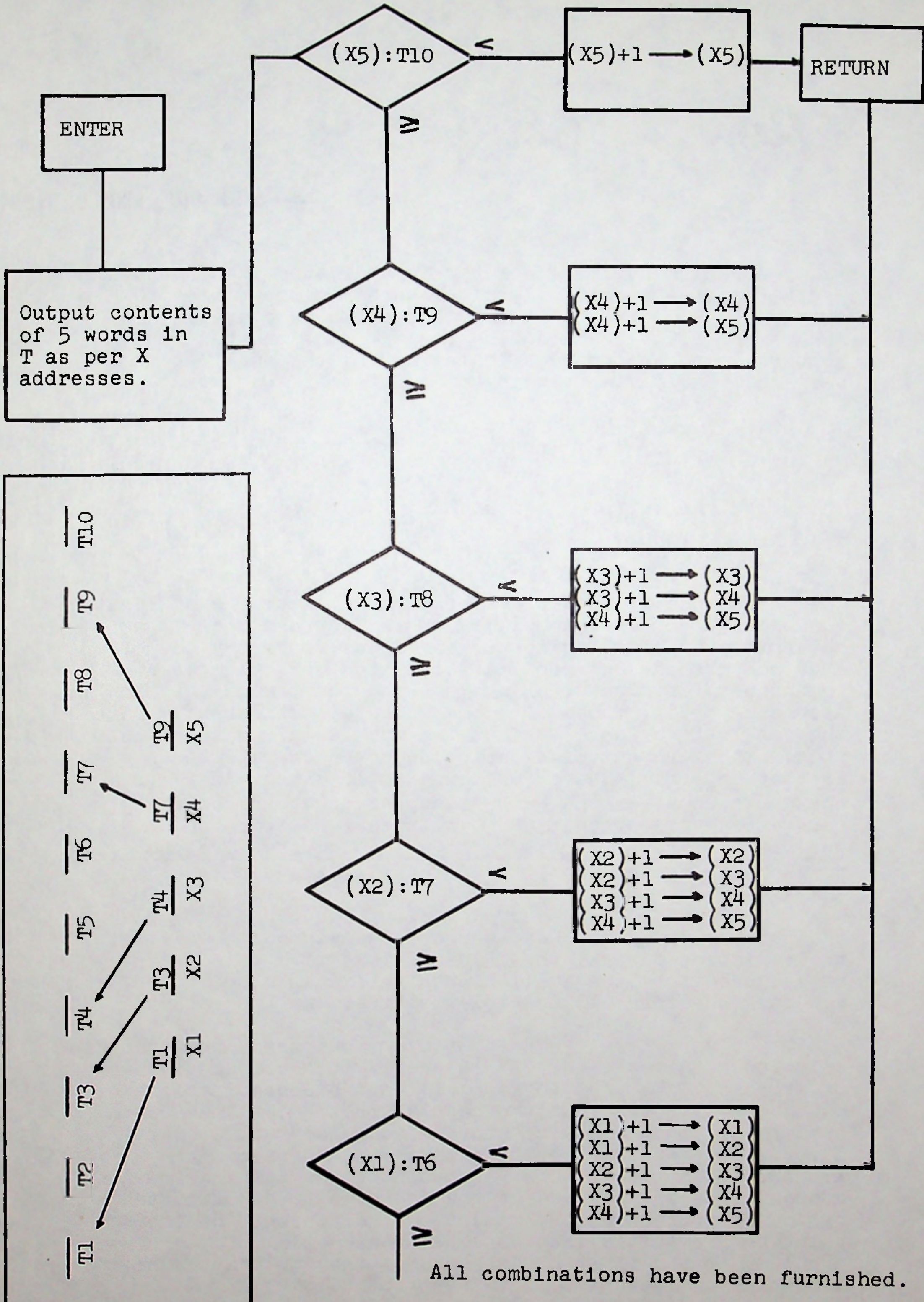
Thus, for the set of numbers:

736	185	572	159	025	673	344	011	427	323
	*		*			*		*	*

the ones marked with a star form the set with the smallest variance (which is 10186.24). This fact could be determined by calculating the variances for every set of five numbers chosen from the set of ten, or by various techniques for cluster analysis (with the ten numbers in ascending order, those five that cluster closest will have the smallest variance). But the game we are playing is this: the five numbers must be accepted or rejected as they appear. Thus, for the example given, there is no way to know, as the numbers appear, that the best cluster centers around 300. With that restriction, what is the proper strategy for selecting the five numbers, to increase the probability of lowering the variance?

Going back to the simpler situation, in which all the numbers are available, suppose one wished to calculate the variances for all the 252 ways that five things can be selected from ten available things? In other words, how does one form all possible combinations from a set?

Let us generalize this problem, but present a specific solution. Given a set of N numbers, in N consecutive words of storage. We want to form all possible combinations of K numbers selected from the population of N. The N numbers are in a block of storage addressed at T1, T2, T3,...,TN. Another set of K words is addressed at X1, X2,...,XK; these words are pointers containing the addresses of the K words of the set of N to be outputted by the subroutine at the moment. The contents of the K pointers are initialized (in the housekeeping of the problem) to the first K addresses of the T block. The accompanying flowchart is in terms of N = 10, K = 5.



All combinations have been furnished.

The boxed picture in the flowchart shows the situation for the 67th set of five to be drawn from the set of ten in block T. The five X pointers contain at the moment the addresses T1, T3, T4, T7, and T9. When that set of elements are outputted, the logic of the flowchart will alter the contents of X5 to be T10. In the flowchart, the actions called for in the rectangles must be taken in order. For example, in the last rectangle, line 1 calls for increasing the contents of X1 by 1, after which that address is to be further incremented by 1 to form the new contents of X2, and so on.

The flowchart, drawn for the specific case of ${}_{10}C_5$, shows a pattern that can be followed for the case ${}_NC_K$.

The generalized procedure forms the basis of our 3rd contest:

FORTRAN : Combinations

A Fortran subroutine is to be written, to output all combinations of K things from an array of N things, one combination per call of the subroutine. The size of the array A (containing the N things) is to be taken as less than or equal to 50. The subroutine, when called, is to put the next K of the N things into an array B. Assume that K is less than or equal to N. If the subroutine is called more than ${}_NC_K$ times, it is then to start over with the first combination.

For the best such subroutine submitted, using only ANSI standard Fortran IV, a prize of $25 is offered. All entries must be received by February 27, 1976. The following are guidelines for the contest:

1. The subroutine should be self-initializing on the first call; that is, the main program should do no initialization other than setting the data values into the main array. The first call of the subroutine should initialize and return the first combination.

2. The subroutine is to be callable by the statement:

```
CALL  COMB(A,B,N,K)
```

where A is a real array of length N; it contains the data values to be combined.
B is a real array of length K; it is the output of the subroutine and contains the next combination.
N is an integer (the length of A)
K is an integer (the length of B)

3. Any other arrays, counters, pointers, etc., must be local to the subroutine.

Entries to Contest Number 3 will be judged on the following criteria:

1. The subroutine must work properly, and the submitted computer printout must show its test procedure.

2. The Fortran logic must be readable and understandable.

3. The subroutine must be documented and COMMENTed. Neatness counts.

4. Between equivalent solutions, weight will be given to the routine that optimizes the tradeoffs between compile time, execution time, and storage space.

Only one prize will be awarded. The decision of the team of judges will be final. The winning program will be published in a subsequent issue of POPULAR COMPUTING. □

PROBLEM 110

Log 32	1.505149978319905976068694473622465133840949407310543
Ln 32	3.465735902799726547086160607290882840377500671801276
$\sqrt{32}$	5.656854249492380195206754896838792314278687501507792
$\sqrt{32}$	3.174802103936398949503411278544616520782986655799706
$\sqrt[10]{32}$	1.414213562373095048801688724209698078569671875376948
$\sqrt[100]{32}$	1.035264923841377504347788194211246197729610910324630
e^{32}	78962960182680.69516097802263510822421995619511535233 06550800205987543078540198889790389126
π^{32}	8105800789910709.653155357989528221108583940555348380 994176096702825129248713537982900155
$\tan^{-1} 32$	1.539556493364628342977609946747260465066089035962153

N-SERIES 32

PRINTED BY
Argold Press, Inc.
16921 VENTURA BOULEVARD / ENCINO, CALIF 91316